Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado en Ingeniería Informática y Matemáticas

Trabajo Fin de Grado

ESTUDIO DE ALGORITMOS DE INTELIGENCIA COMPUTACIONAL BASADOS EN ENJAMBRES

Autor: Mateo Fernández de Gamboa Santiuste
Tutor: David Camacho Fernández

Enero 2018

ESTUDIO DE ALGORITMOS DE INTELIGENCIA COMPUTACIONAL BASADOS EN ENJAMBRES

Autor: Mateo Fernández de Gamboa Santiuste
Tutor: David Camacho Fernández

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero 2018

Resumen

Resumen

En la naturaleza encontramos situaciones y procesos para los que no se encuentra una explicación inmediata. Un caso particular de esto son los enjambres, en los que muchos individuos simples actúan de manera que emerge un comportamiento inteligente. La inteligencia de enjambre es la rama de la inteligencia artificial que busca estudiar y simular los complejos comportamientos encontrados en estos enjambres, como por ejemplo las bandadas de aves o las colonias de hormigas.

En este caso, se implementará un algoritmo basado en el comportamiento de los machos de una rana japonesa, aplicándolo al problema de coloreado de grafos. Este algoritmo se llamará *FrogCol*. Aparte se ha implementado una versión de la metaheurística ACO aplicada a este mismo tipo de problema, *AntCol*, para tener otro algoritmo con el que comparar los resultados obtenidos. Tras estudiar la estructura de estos y otros algoritmos de enjambre, se han abstraído sus funcionalidades básicas, creando una estructura de clases robusta y fácilmente ampliable. Esta estructura servirá para facilitar el uso de distintos tipos de algoritmos de enjambre orientados a colorear grafos, un problema de interés actual por sus aplicaciones en materias tan distintas como resolver *sudokus* o repartir horarios de trabajo.

Para analizar la eficiencia y viabilidad de los algoritmos implementados, se ha decidido implementar un generador de grafos geométricos aleatorios usando *NetworkX*. Esto ha permitido organizar una batería de pruebas con grafos de distinto tamaño y densidad, obteniendo muchos datos interesantes en el proceso. Analizando los resultados, se concluye que el algoritmo *FrogCol* es muy eficiente, y obtiene soluciones mejores en menos tiempo que *AntCol*. Viendo este resultado tan positivo, se proponen nuevos desarrollos de interés para realizar en el futuro.

Palabras Clave

Inteligencia Computacional, Inteligencia de Enjambre, Algoritmos de Enjambre, Optimización por Colonia de Hormigas, Coloreado por Ranas, Coloreado de Grafos, Programación Orientada a Objetos

Abstract

In nature, there are processes and situations for which we have a hard time finding an immediate explanation. In particular, swarms are systems in which simple individuals act in such a way that an intelligent behavior emerges. Swarm intelligence is the branch of artificial intelligence that studies and tries to simulate the complex behaviors commonly found in swarms, such as bird flocking or ant colonies.

In this case, an algorithm based on the behavior of male specimens of a Japanese frog is going to be implemented, applying it to graph coloring. This algorithm is named *FrogCol*. Apart from this, an adaptation of the ACO metaheuristic for this problem, *AntCol*, has been implemented, in order to have a different algorithm with which to compare the results obtained. After studying their and other swarm algorithms' structure, their basic functionalities have been generalized, creating a robust and easily extended class framework. This framework will be used to simplify the use of different graph-coloring-oriented swarm algorithms, a problem of interest due to its many different applications, such as solving *sudokus* or work scheduling.

To analyze the efficiency and viability of the implemented algorithms, it has been decided to code a random geometric graph generator using *NetworkX*. This has allowed the creation of a test suite, using graphs of different sizes and densities which, in turn, provide many interesting data. Analyzing the results, it has been concluded that the *FrogCol* algorithm is very efficient and obtains better results in less time than *AntCol*. Looking at such positive results, several interesting future developments are proposed.

Key words

Computational Intelligence, Swarm Intelligence, Swarm Algorithm, Ant Colony Optimization, Frog Coloring, Graph Coloring, Object Oriented Programming

Agradecimientos

Gracias a David, por su paciencia y entusiasmo, cuya energía me convenció de hacer este trabajo.

Gracias a Irene y Javi, por hacer del despacho una segunda casa aunque las cosas no acabaran bien.

Gracias al resto del equipo de AIDA, con el que he compartido unos meses muy importantes de mi vida.

Gracias a mis amigos y amigas, que me han levantado el ánimo siempre que no creía que fuese a acabar.

Gracias a Pinar, por estar ahí siempre que la he necesitado. Sin ella no podría haber hecho todo esto.

Y por supuesto, gracias a mis padres, por acompañarme desde el principio.

Índice general

Índice de Figuras IX

Índice de Tablas X

1. Introducción 1

1.1. Motivación . . . 1

1.2. Objetivos y enfoque . . . 2

1.3. Estructura del documento . . . 2

2. Estado del arte 5

2.1. Inteligencia de enjambre . . . 5

2.1.1. Conceptos básicos . . . 5

2.1.2. Modelos . . . 7

2.1.3. Metaheurísticas y algoritmos . . . 7

2.2. Teoría de grafos . . . 9

2.2.1. Conceptos básicos . . . 9

2.2.2. Problema de coloreado de grafos . . . 10

3. Sistema, diseño y desarrollo 13

3.1. Algoritmos . . . 13

3.1.1. FrogCol . . . 13

3.1.2. AntCol . . . 15

3.2. Estructura de la implementación . . . 16

3.3. Creación de grafos . . . 19

3.3.1. Grafos geométricos aleatorios . . . 19

3.3.2. *NetworkX* . . . 19

4. Experimentos realizados y resultados 21

4.1. *FrogCol* . . . 22

4.1.1. Tiempos . . . 22

4.1.2. Coloreados . . . 23

4.1.3. Grupos maximales independientes 24
4.2. *AntCol* 25
4.2.1. Tiempos 25
4.2.2. Coloreados 26
4.2.3. Grupos maximales independientes 26
4.2.4. Número de agentes 27
4.3. Resultados del experimento 28

5. Conclusiones y trabajo futuro **29**
5.1. Conclusiones 29
5.2. Trabajo futuro 30

Glosario de acrónimos **31**

Bibliografía **32**

A. Código Implementado **35**
A.1. Fichero base.py 35
A.1.1. Node 35
A.1.2. Colony 35
A.1.3. Agent 36
A.2. Fichero frogcol.py 37
A.2.1. FrogColony 37
A.2.2. FrogAgent 39
A.2.3. FrogMessage 40
A.3. Fichero antcol.py 40
A.3.1. AntColony 40
A.3.2. AntAgent 42
A.4. Fichero graphgen.py 46

Índice de Figuras

2.1. El comportamiento de las abejas es un ejemplo de enjambre en la naturaleza . . 6
2.2. El algoritmo PSO fue diseñado en primera instancia para simular el comportamiento de las bandadas de pájaros o los bancos de peces 9
2.3. Coloreado de vértices del grafo de Petersen 10

3.1. Diagrama de clases 18
3.2. Ejemplo de grafo geométrico aleatorio creado con *NetworkX* 19

4.1. Representación de tiempo empleado por *FrogCol* respecto a nº de nodos, $r = 0{,}1$ 22
4.2. Representación de tiempo empleado por *AntCol* respecto a nº de nodos, $r = 0{,}1$. 25
4.3. Comparativa de coloreados y MIS obtenidos por *FrogCol* y *AntCol* 28

Índice de Tablas

4.1. Tabla de tiempos de ejecución de *FrogCol* 22
4.2. Tabla de coloreados de *FrogCol* 23
4.3. Tabla de obtención de MIS de *FrogCol* 24
4.4. Tabla de tiempos de ejecución de *AntCol* 25
4.5. Tabla de coloreados de *AntCol* 26
4.6. Tabla de obtención de MIS de *AntCol* 26
4.7. Tabla comparativa de *AntCol* con distinta cantidad de agentes 27

1 Introducción

La naturaleza siempre ha servido de inspiración al ser humano, siendo objeto de todo tipo de estudios e investigaciones. En el campo de la inteligencia artificial esto se puede ver reflejado de muchas fomas. Por ejemplo, los algoritmos genéticos están basados en el proceso de evolución de las cadenas de ADN, y las redes neuronales artificiales buscan imitar el funcionamiento del cerebro. Otra de las ramas de estudio de la inteligencia artificial es la llamada inteligencia de enjambre, expresión cuyo origen se puede trazar a Beni y Jing [1].

La inteligencia de enjambre es la rama de la inteligencia artificial que estudia el comportamiento colectivo de los sistemas descentralizados, aquellos que están basados en poblaciones de elementos simples que interactúan entre sí y con el entorno. Las interacciones dan lugar a un comportamiento global *inteligente* a pesar de que los agentes de forma individual siguen normas simples. A partir del estudio de estos sistemas se desarrollan multitud de algoritmos, aplicables a problemas como el coloreado de grafos o el problema del viajante, entre otros.

El coloreado de grafos en particular es un subgrupo de problemas dentro del etiquetado de grafos. Ha sido estudiado como problema algorítmico desde 1970, aunque fue planteado por primera vez al adaptar el problema del coloreado de mapas. Consiste en asignar un color a cada vértice, de forma que dos vértices vecinos no puedan tener el mismo asignado. Este problema es de complejidad NP-completo, por lo que se han usado algoritmos derivados de la inteligencia de enjambre para aproximar una solución óptima en un espacio de tiempo razonable. Algunas de sus aplicaciones son la asignación de horarios o la asignación de radio frecuencias.

1.1. Motivación

El problema del coloreado de grafos ha sido estudiado durante mucho tiempo, pero al ser de dificultad NP-completo resulta imposible implementar un método fijo para encontrar la solución óptima. La aplicación de heurísticas y algoritmos de aproximación nos permiten hayar una solución factible, aunque no sea óptima. Este trabajo se origina para hacer un estudio de un algoritmo de inteligencia de enjambre de reciente creación, *FrogCol* [2], aplicado al coloreado de grafos.

La inteligencia de enjambre ha obtenido críticas en los últimos tiempos debido al aparente estancamiento en cuanto a innovación en este área. Muchos algoritmos son publicados y publicitados sin un correcto rigor científico, basándose en nuevas metáforas y complejas nomenclaturas

que hacen difícil su comparación con otros desarrollos de la literatura. Esto oculta muchas veces un notable parecido con estudios pasados, haciendo que la validez del estudio caiga en entredicho. Sin embargo, este algoritmo contiene características innovadoras, que se demostrarán al ser explicado extensivamente en el documento.

Para poder valorar correctamente el funcionamiento y la eficiencia de FrogCol, se ha buscado una adaptación de un algoritmo de referencia como el ACO para el problema sobre el que se realizarán las pruebas. De esta forma, se ejecutarán ambos algoritmos para poder comparar sus resultados y valorar correctamente a *FrogCol*. Se compararán los tiempos y los coloreados mínimos, además de la búsqueda de grupos máximos independientes, MIS (*maximal independent set*), de forma indirecta. Esto se realizará sobre *python*, lenguaje potente con gran cantidad de paquetes adecuados para entornos científicos y uno de los que más crece en los últimos años [3].

1.2. Objetivos y enfoque

Este trabajo se centra en un algoritmo de reciente desarrollo, perteneciente al campo de la inteligencia de enjambre. Este algoritmo es el denominado como *FrogCol*, y fue descrito por primera vez por Hugo Hernández y Christian Blum [2]. Fue implementado inicialmente para optimizar el coloreado de grafos, y después adaptado para la búsqueda de un MIS dentro de un grafo [4]. A partir de esto, los objetivos de este trabajo son:

1. Implementación de superclases. Una de las ideas secundarias de este proyecto consistía en abstraer las cualidades de los algoritmos de enjambre, de forma que se construya una estructura de clases bajo el paradigma OOP, Programación Orientada a Objetos.

2. Implementación de algoritmo *FrogCol*. Basándose en el código en *C++* de Christian Blum, se implementará el algoritmo *FrogCol* ligeramente simplificado en *python*. Además, se juntará con *FrogMIS*, extrayendo también el subgrupo maximal independiente del grafo.

3. Implementación de algoritmo *AntCol*. Tomando como referencia el pseudocódigo presentado por Costa y Hertz en 1997 [5], se implementará *AntCol* en *python*. Además, se añadirá la detección de MIS de forma indirecta, usando un método análogo al usado en *FrogCol*.

4. Generación de grafos geométricos aleatorios mediante *NetworkX*. Los grafos que vamos a usar para probar los algoritmos serán generados mediante esta librería. Estos grafos son no dirigidos, y están construidos determinando únicamente el número de nodos y un radio, que determina los enlaces que se crearán.

5. Aplicación de los algoritmos y estudio de los resultados generados. Una vez generados los grafos y completados los algoritmos, se aplicarán sobre los grafos con variaciones en cuanto al número de iteraciones y otros parámetros. Estos resultados se compararán y se analizarán.

1.3. Estructura del documento

El documento está dividido en las siguientes secciones:

- **1. Introducción:** en esta primera sección se hace una breve presentación de la inteligencia de enjambre y sus orígenes inspirados en la naturaleza. Después, se menciona el problema de coloreado de grafos, uno de los problemas donde típicamente se han aplicado los algoritmos de enjambre.

- **2. Estado del Arte:** la primera parte es una explicación de conceptos propios de la inteligencia de enjambre, seguida de los primeros modelos de los enjambres y los más importantes desarrollos modernos en cuanto a metaheurísticas y algoritmos. Después se explican algunos conceptos básicos de la teoría de grafos y, por último, una descripción del GCP y los métodos que existen para abordarlo.
- **3. Sistema, Diseño y Desarrollo:** en esta sección se explican los algoritmos implementados y la estructura con la que se ha hecho. También se muestra la forma en la que se han creado los grafos de prueba y la herramienta con la que se ha hecho.
- **4. Experimentos:** se detallan las pruebas realizadas con los algoritmos ya implementados, además de presentar los resultados obtenidos.
- **5. Conclusiones y Trabajo Futuro:** esta última sección contiene las conclusiones extraídas del desarrollo de los algoritmos y de los resultados obtenidos. La última sección del documento detalla posibles trabajos futuros a tener en cuenta.

2

Estado del arte

2.1. Inteligencia de enjambre

Como hemos visto en la introducción, la inteligencia de enjambre es una rama de estudio que se centra en analizar el comportamiento colectivo de sistemas descentralizados. En ellos se observan comportamientos útiles que están más allá de lo que cada uno de los individuos del enjambre sería capaz de realizar. Los elementos que componen estos enjambres se suelen denominar *agentes*, y cada uno de ellos tiene un comportamiento relativamente simple. Es la interacción entre los propios agentes y el entorno lo que hace que emerja un comportamiento inteligente.

Este área esta siendo objeto de mucho estudio en la actualidad [6], gracias en parte a sus posibles aplicaciones en problemas físicos. El hecho de no necesitar complejos controladores centrales tendría como consecuencia que los sistemas fuesen más robustos a posibles averías o daños, además de muy posiblemente ver su coste rebajado. Además, debido a sus propiedades, resulta útil modelar y realizar estudios de inteligencia de enjambre para entender mejor los enjambres presentes en la naturaleza, como las colonias de hormigas o enjambres de abejas.

2.1.1. Conceptos básicos

La inteligencia de enjambre está basada en varios conceptos interdisciplinares, cuyo origen en muchos casos se puede trazar hasta la naturaleza.

Definición 2.1.1.1 *La **emergencia** es el fenómeno por el cuál aparecen propiedades o procesos en un sistema que no se pueden atribuir a los elementos individuales que componen el sistema. Son propiedades no reducibles a las de las partes constituyentes.*

Este concepto esta relacionado estrechamente con el concepto de autoorganización.

Definición 2.1.1.2 *La **autoorganización** es un proceso en el que interacciones locales entre los componentes de un sistema dan lugar a un orden o coordinación global. Es espontáneo y no está dirigido ni controlado por ningún elemento en particular.*

El resultado sobre el que se centra el estudio de la inteligencia de enjambre es, como dice su nombre, el enjambre. Es un ejemplo particular de emergencia y autoorganización que se da en la naturaleza.

Definición 2.1.1.3 *Un **enjambre** o el **comportamiento de enjambre** es un comportamiento emergente que surge a partir de reglas simples seguidas por una serie de individuos y que no requiere una coordinación central. Es un comportamiento colectivo exhibido por entidades similares al estar en grupo. El término enjambre se usa de forma usual para insectos, pero se aplica para cualquier entidad que muestre comportamiento de enjambre.*

Figura 2.1: El comportamiento de las abejas es un ejemplo de enjambre en la naturaleza

Debido a las propiedades de los enjambres, resulta más sencillo fijarse y concretar las reglas básicas que sigue cada organismo del enjambre que centrarse en el comportamiento global.

Definición 2.1.1.4 *Un **modelo basado en agentes** es una metodología basada en reglas de modelización computacional. Se centra en las reglas e interacciones de los componentes individuales o agentes de un sistema. Posee ciertas características:*

a) Estructura modular: el comportamiento del modelo se define mediante las reglas de sus agentes. Estas pueden modificarse sin modificar el modelo al completo.

b) Propiedades emergentes: al usar agentes que interactúan localmente, se consigue un sistema con un comportamiento mucho más complejo que el de cada agente individual.

c) Abstracción: los modelos basados en agentes pueden construirse aún sin tener el conocimiento completo del sistema que se va a estudiar o representar.

d) Estocasticidad: los sistemas biológicos poseen comportamientos que parecen aleatorios. Para simular esto, se calculan las probabilidades de los comportamientos y se traducen en las reglas de los agentes.

Para entender los modelos basado en agentes, es necesario especificar cómo son los agentes que componen los sistemas simulados.

Definición 2.1.1.5 *Un **agente inteligente** es una entidad autónoma que posee sensores para percibir su entorno y actúa sobre él con actuadores, de forma que su actividad le ayude a*

conseguir cierto objetivo. En el caso computacional, esto significa que debe existir un sistema por el que el agente pueda comunicarse o actuar sobre un espacio global que comparte con los demás agentes o estos agentes en sí. Sus acciones estarán definidas o condicionadas por este mismo entorno.

Definición 2.1.1.6 *La **estigmergía** es un mecanismo de autoorganización en el que se consigue coordinación entre elementos de un sistema a través del entorno. Se basa en que las trazas dejadas en el entorno por un agente estimulan la eficiencia de la siguiente acción, ya sea del mismo agente o de otro. Produce estructuras complejas y aparentemente inteligentes, sin ser necesario un plan, control o, incluso, comunicación directa entre los agentes.*

Este mecanismo se usa junto a agentes muy simples, que pueden carecer de memoria o incluso conciencia de la existencia de otros agentes. De esta forma, el entorno sería usado como una forma de memoria colectiva, a la que todos los agentes tienen acceso. En la naturaleza, se puede observar en el funcionamiento de las colonias de hormigas o termitas por ejemplo, que usan feromonas para optimizar sus trabajos.

La combinación de estas propiedades se utiliza para buscar soluciones a problemas cuya complejidad hace que el tiempo necesario para hallar la solución óptima crezca demasiado rápido. De esta forma, se especifican ciertos criterios que las soluciones candidatas deben tener y se busca la mejor posible en cierto tiempo, obteniendo resultados aproximados pero buenos. Los procesos que llevan a cabo esto se denominan metaherísticas.

Definición 2.1.1.7 *Una **metaheurística** es un proceso o heurística diseñado para encontrar o generar un algoritmo de búsqueda que proporcione una solución lo suficientemente buena a un problema de optimización. No garantizan que se alcance una solución óptima global, si no que implementan algún tipo de optimización estocástica. Así, la solución encontrada depende del grupo de variables aleatorias generadas.*

2.1.2. Modelos

En las últimas décadas se han desarrollado diversos tipos de modelos para estudiar el comportamiento de los enjambres. Esta tendencia se debe a que al modelar el sistema se pueden llegar a entender mejor los enjambres y su funcionamiento en la naturaleza, encontrándose nuevas aplicaciones para ellos.

Los primeros estudios en este campo usaban modelos matemáticos para simular y entender los comportamientos de los enjambres. La primera simulación de este comportamiento en un ordenador se realizó en 1986, con el programa *boids* [7]. Toma el nombre de la contracción de *bird-oid object*, referente a su intención de simular el comportamiento de las bandadas de pájaros. Este modelo se ha ido extendiendo de diversas formas a lo largo del tiempo [8, 9], dando lugar a programas más especializados. A día de hoy se sigue utilizando, por ejemplo, en el campo de la robótica y los drones [10].

2.1.3. Metaheurísticas y algoritmos

En el estudio de la inteligencia de enjambre se ha recurrido a fenómenos que ocurren en la naturaleza para elaborar procesos que, imitándolos, puedan dar solución a ciertos tipos de problemas. Estos algoritmos son metaheurísticas, ya que obtienen a partir de el resultado emergente soluciones factibles, sin garantizar una solución óptima global. Aún así, debido a que se

aplican a problemas de alta complejidad generalmente, estas soluciones se encuentran mucho más rápida y eficientemente que si se usara un algoritmo de optimización global.

Existen muchos tipos de algoritmos de enjambre, aunque los más conocidos y utilizados son los conocidos como PSO, optimización por enjambre de partículas, y ACO, optimización por colonia de hormigas. Cabe señalar que algunas metaheurísticas más recientes basadas en fenómenos naturales han sido criticadas por esconder su falta de innovación detrás de una elaborada metáfora [11, 12].

Optimización por Colonia de Hormigas

Inicialmente propuesto por Dorigo en 1992 en su tesis doctoral [13, 14], es una clase de algoritmos de optimización basada en el comportamiento de una colonia de hormigas. Las hormigas buscan comida, depositando feromonas en el camino de vuelta. Esto afecta al resto de amigas, que son más proclives a elegir un camino con mayor cantidad de feromonas. Computacionalmente, las hormigas son agentes simulados que se mueven por el espacio de soluciones, marcando sus recorridos y su calidad. Esto atrae a los demás agentes, de forma que en las iteraciones siguientes se encuentren mejores soluciones. Este tipo de comunicación se basa en el mecanismo de estigmergía, ya que usa el entorno como medio.

Algoritmo 1: Pseudocódigo de metaheurística ACO

```
Initialize pheromone τ_i
while not termination:
    for each ant i:
        Construct solution i
        Update pheromone τ_i
```

El primer algoritmo desarrollado con este planteamiento estaba pensado para encontrar un camino en un grafo, lo que se conoce como TPS, el problema del viajante (*Travelling Salesman Problem*). Desde entonces, la idea se ha ido diversificando para resolver otros tipos de problemas, como el coloreado de grafos [5], descrito en la sección 3.1.2, o la clasificación de datos [15].

Optimización por Enjambre de Partículas

Propuesto por Kennedy y Eberhart en 1995 [16], el PSO es un método computacional que trata de optimizar un problema intentando mejorar iterativamente un candidato a solución respecto a una medida de calidad. Representa los candidatos a solución mediante partículas en el espacio de búsqueda, aplicándolesn una velocidad. Las partículas son atraídas por la mejor posición local, pero también a las posiciones de otras partículas dependiendo de lo buenas que sean las soluciones encontradas.

Algoritmo 2: Pseudocódigo de metaheurística PSO [17]

```
Initialize x_i, v_i and xbest_i for each particle i
while not termination:
    for each particle i:
        Evaluate objective function
        Update xbest_i
    for each i:
        Set g equal to index of neighbor with best xbest_i
        Use g to calculate v_i
        Update x_i = x_i + v_i
        Evaluate objective function
        Update xbest_i
```

Figura 2.2: El algoritmo PSO fue diseñado en primera instancia para simular el comportamiento de las bandadas de pájaros o los bancos de peces

El algoritmo estaba dirigido originalmente a simular el comportamiento social, basado en el movimiento de organismos en una bandada de pájaros o un banco de peces. A partir de ahí el algoritmo fue simplificado y se observó la posibilidad de usarse en optimización de una forma más general. Hoy en día se usa en multitud de áreas, desde el diseño de antenas hasta asignación de trabajo [18].

Una lista de metaheurísticas y algoritmos ha sido compilada por Fister *et al* [19]. De una manera menos seria, también existe un *bestiario* mucho más extenso, mantenido por Aranha y Campelo y publicado en *github* [20].

2.2. Teoría de grafos

La teoría de grafos es el área de las matemáticas dedicada al estudio de los grafos, estructuras formadas por un grupo de objetos entre los cuales existen relaciones de algún tipo. El nombre grafo fue usado por primera vez con este significado por Sylvester en 1878 [21]. Para el estudio de algoritmos de enjambre llevado a cabo en este proyecto se va a usar este área de conocimiento, ya que se van a centrar en encontrar soluciones al problema de coloreado de grafos.

2.2.1. Conceptos básicos

En esta sección se van a definir algunos conceptos pertenecientes a la teoría de grafos, necesarios para entender el problema tratado en este trabajo. También se usarán en el funcionamiento de los algoritmos implementados, valiéndose de ellos para encontrar candidatos a solución.

Definición 2.2.1.1 *Un **grafo** es un sistema de nodos o vértices, conectados entre sí por enlaces o arcos. Es el objeto de estudio de la Teoría de Grafos.*

Los grafos se han dividido comúnmente en dos tipos, dependiendo de si sus enlaces poseen una dirección definida, carecen de ella, o existen ambas posibilidades.

Definición 2.2.1.2 *Un grafo **dirigido** es aquél en el que los ejes tienen una dirección particular, con un nodo origen y un nodo destino. Un grafo **no dirigido**, sin embargo, es aquél en el que los enlaces no distinguen entre origen y destino, no tienen dirección. Un grafo **mixto** posee enlaces de los dos tipos.*

Centrándonos ahora en los nodos de los grafos, se pueden estudiar algunas de sus propiedades basándonos en la estructura del grafo al que pertenecen.

Definición 2.2.1.3 *Un **vecino** de un nodo es otro nodo que está conectado a él por un enlace, es decir, un nodo adyacente. La **vecindad** o **neighborhood** de un nodo v en un grafo W es el subgrafo compuesto por todos los nodos adyacentes a v, y está representado por $N_W(v)$.*

Definición 2.2.1.4 *El **grado** de un nodo v en un grafo W es el número de enlaces que terminan en él. En el caso de un grafo no dirigido, es directamente el número de enlaces que salen o entran en el nodo. Está representado por $deg_W(v)$.*

Definición 2.2.1.5 *Un **camino** entre dos nodos es una secuencia de nodos y enlaces que empieza en uno de los nodos y acaba en el otro.*

2.2.2. Problema de coloreado de grafos

En teoría de grafos, el coloreado de grafos es un caso especial de etiquetado de grafos. Consiste en asignar una serie de etiquetas llamadas colores a cada nodo, teniendo distintas restricciones dependiendo del tipo de coloreado que se quiera hacer. Este concepto de colorear grafos proviene del coloreado de mapas, en el que distintos territorios debían ser coloreados siendo necesario que territorios adyacentes tomasen colores distintos. Es así como se define el tipo de coloreado de grafos más simple, el **coloreado de vértices**, en el que se debe colorear un grafo de forma que ningún par de vértices adyacente posea el mismo color.

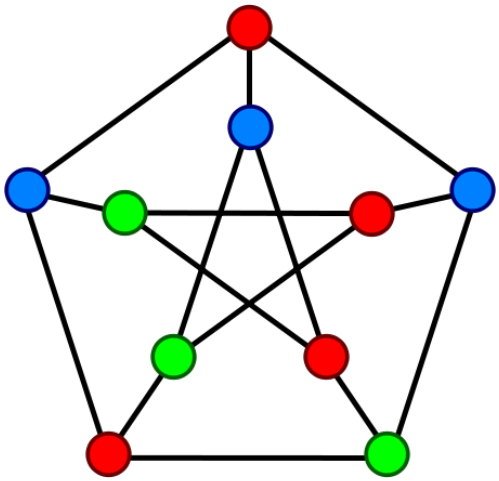

Figura 2.3: Coloreado de vértices del grafo de Petersen

Una de las nociones más importantes del coloreado de vértices es el **Teorema de los Cuatro Colores**. Fue mencionado por primera vez por Francis Guthrie en 1852 [22], pero no fue hasta 1976 cuando Appel y Haken probaron el teorema en la primera gran demostración usando un ordenador [23]. Este teorema dice que, dada cualquier separación de un plano en regiones contiguas, no se necesitan más de cuatro colores para que ningún par de regiones adyacentes

tenga el mismo color. Esto se puede extrapolar a los grafos planares, es decir, aquellos que pueden ser representados en un plano sin que sus enlaces se corten entre sí. Para grafos no planares, tenemos el **Teorema de Brooks**. Este nos dice que un grafo conexo en el que el máximo grado de un nodo es Δ, puede ser coloreado con sólo Δ colores. Hay dos excepciones, grafos completos y grafos cíclicos de orden impar requieren $\Delta + 1$ colores [24].

El problema de coloreado de grafos, GCP por *Graph Coloring Problem*, consiste en encontrar el menor número posible de colores con el que se pueda hacer un coloreado de vértices de un grafo dado, lo que se denomina el **número cromático**. Este problema se encuentra entre los 21 problemas NP-completos de Karp [25], por lo que no se ha descubierto ningún método para computar su solución en un periodo razonable de tiempo. Para aproximar estas soluciones, se han usado métodos heurísticos y algoritmos de aproximación como el algoritmo de contracción [26], algoritmos codiciosos [27] o algoritmos descentralizados [28, 29]. En nuestro caso, usaremos algoritmos de enjambre basados en colonias de ranas y hormigas, *FrogCol* y *AntCol*.

3

Sistema, diseño y desarrollo

En esta sección se describe el funcionamiento de los algoritmos *FrogCol* y *AntCol* y la forma en la que se han implementado. Después, se explica el sistema de clases desarrollado para usar los algoritmos. También se añaden comentarios sobre cómo se han generado los grafos sobre los que se han probado los algoritmos y sus propiedades.

3.1. Algoritmos

Este proyecto se ha centrado en un primer momento en la implementación en *python* y la aplicación del algoritmo *FrogCol*, definido en primera instancia por Hernández y Blum en 2010 [2]. Este algoritmo se centra en la búsqueda de un coloreado óptimo de un grafo. Para que los resultados puedan ser comparados con un algoritmo del estado del arte, se ha decidido implementar una versión del ACO aplicada al mismo problema. La metaheurística ACO ha sido objeto de muchos estudios e implementaciones a lo largo de los años, por lo que es un estándar con el que se puede comparar y obtener conclusiones útiles. Esta implementación está basada en el algoritmo *AntCol* descrito por Costa y Hertz en 1997 [5], realizándose en *python* a partir del pseudocódigo publicado. Ambos algoritmos pertenecen al campo de la inteligencia de enjambre, aunque el enfoque del problema es muy distinto. Por todo esto, resulta interesante averiguar de qué forma se comparan, tanto en cuanto a tiempo empleado como a calidad de los resultados.

Una vez planteado el estudio, se han encontrado similitudes en los procesos desarrollados por los dos algoritmos. Esto es normal en algoritmos y heurísticas derivados de la inteligencia de enjambre, pero en este caso se vio la posibilidad de abstraer su funcionalidad, creando una estructura de clases jerárquica que fuese fácil de ampliar en el futuro. Esta estructura se describirá en el apartado 3.2.

3.1.1. FrogCol

El algoritmo *FrogCol* tiene su origen en el comportamiento de una rana arborícola japonesa. Los individuos macho de esta especie deben croar para llamar la atención de la hembra, pero ésta no puede distinguir de dónde provienen las llamadas si varias ranas cercanas croan a la vez [30]. Por ello, han desarrollado un sistema de desincronización, en el que las ranas macho

escuchan a otras ranas cercanas, cambiando la frecuencia de sus llamadas para que no colisionen y confundan a la hembra. Tras varias iteraciones del proceso, los machos que ocupan posiciones cercanas se han desincronizado totalmente.

A partir de este comportamiento, Aihara propuso en 2009 [31] un modelo en el que una pareja de osciladores, representando cada uno una rana macho, buscan tener fases distintas mediante una desincronización gradual. Hernández y Blum diseñaron en 2010 el algoritmo *FrogCol* [2], el cual, con ligeras modificaciones, será el algoritmo que se implementará en *python* y se estudiará. Es una adaptación del modelo basado en las ranas, destinado a optimizar el GCP.

Algoritmo 3: Pseudocódigo de *FrogCol*

```
f_0 := ∞   /* Número de colores en la mejor solución obtenida hasta ahora */
n_0 := ∞          /* Número de repeticiones máximo de un color hasta ahora */
Sort agents by θ
while not termination:
    for each agent i:
        θ_i := calculateNewThetaValue()
        c_i := minimumColorNotUsed()
        sendColoringMessage()
        clearMessageQueue()
    q = colorsUsed()
    if q < f_0:
        solution := currentColoring
        f_0 := q
    n_aux = maxRepColor()
    if n_aux > n^0:
        n_0 := n_aux
    Sort agents by θ
```

Este algoritmo comienza asignando a cada nodo un agente, que simulará una rana. Estos agentes son generados con un valor θ aleatorio, que se usará después para estructurar el grafo en forma de árbol. Una vez ordenados, los agentes pasan a ejecutar en secuencia su evento, que en el caso de *FrogCol* consta de varias partes. Primero se recalcula su nuevo valor θ, basado en los valores encontrados en la cola de mensajes Q_i. Estos mensajes incluyen el valor θ_m y el color del nodo emisor. Así, la fórmula del nuevo θ_i para el agente del nodo i es:

$$\theta_i := \theta_i - \alpha \sum_{m \in Q_i} \frac{\sin(2\pi(\theta_m - \theta_i))}{2\pi} \tag{3.1}$$

donde $\alpha \in [0, 1]$ es un parámetro usado para controlar la convergencia. Después, el nodo escoge el mínimo color posible comparando con los colores presentes en los mensajes almacenados en Q_i. Una vez ha escogido el color, se ha actualiza el número de colores usado, almacenado en q, y el agente comunica a todos los adyacentes su color. Esto lo hace mediante un mensaje que incluye también su valor θ.

Una vez todos los agentes se han comunicado, se extrae un candidato a solución del GCP. Este proceso se repite varias veces, quedándose al final con la mejor solución, es decir, la solución que menos colores haya usado para colorear el grafo. También se obtiene un candidato a MIS, guardando la coloración con la mayor repetición de un mismo color. Cabe destacar que este algoritmo de enjambre no hace uso del concepto de estigmergía, ya que toda la comunicación se hace directamente entre los propios agentes, sin usar el entorno como medio de comunicación y almacenamiento de información.

3.1.2. AntCol

El algoritmo *AntCol* es una implementación de la metaheurística ACO, que modela el comportamiento de una colonia de hormigas, para el ámbito del coloreado de grafos. Para este proyecto se va a implementar una versión del algoritmo definido por Costa y Hertz en 1997 [5] en *python*.

En su artículo, definen el algoritmo en dos partes. La primera parte consiste en el mantenimiento y actualización de M_{rs}, la matriz que representa los rastros de feromonas que dejan las hormigas al ir en busca de comida. En este caso cada agente recorre el grafo coloreando cada vez con un color distinto, así que los elementos de la matriz son proporcionales a la calidad de los coloreados en los que los nodos r y s poseen el mismo color. El agente ordena los nodos del grafo para poder recorrerlo, y es esta construcción la que determina la calidad de la solución obtenida. El método constructivo encargado de ordenar el grafo tiene en cuenta el estado de la matriz de rastros, dando más importancia a los caminos con enlaces más transitados por los otros agentes.

Algoritmo 4: Pseudocódigo de *AntCol*

```
M_rs := initializeMatrix(1)     /* Matriz de rastros entre nodos no adyacentes */
f^0 := ∞   /* Número de colores en la mejor solución obtenida hasta ahora */
n_0 := ∞          /* Número de repeticiones máximo de un color hasta ahora */
while not termination:
    ΔM_rs := initializeMatrix(0)
    for each agent i:
        ANT_RLF(2,2)
        q := colorsUsed()
        if q < f^0:
            solution := currentColoring
            f^0 := q
        n_aux := maxRepColor()
        if n_aux > n^0:
            n_0 := n_aux
        ΔM_rs := updateTrail(ΔM_rs)
    M_rs := ρM_rs + ΔM_rs∀[v_r, v_s]non adjacent
```

La actualización de la matriz de rastros se hace al final de cada iteración, una vez todos los agentes han conseguido un candidato a solución. Para ello se define una matriz ΔM que acumula los cambios que se van a realizar en la matriz de rastros M. Tras la ejecución del evento de cada agente ΔM se ve modificada de la siguiente manera:

$$\Delta M_{rs} = \Delta M_{rs} + \sum_{S_a \in Srs} \frac{1}{q_a} \tag{3.2}$$

Costa y Hertz proponen y analizan diversas formas de ordenar el grafo. Viendo sus conclusiones y los resultados obtenidos, se ha decidido implementar el método constructivo *ANT_RLF*, *Recursive Largest First*, con parámetros $\Sigma = 2, \Omega = 2$. El primer parámetro, Σ, define que, con cada color usado, el primer nodo se selecciona aleatoriamente entre los nodos sin colorear. El segundo parámetro de *ANT_RLF* determina el cálculo de $\eta(s[k-1], v)$. Esta función es una parte del cálculo de la probabilidad $p_{it}(k, v)$, que es la probabilidad asignada a cada nodo y determina cuál de ellos se elegirá para ser coloreado a continuación. La expresión completa de la función de probabilidad es la siguiente:

$$p_{it}(k,v) = \frac{\tau(s[k-1],v,q)^{\alpha} \cdot \eta(s[k-1],i)^{\beta}}{\sum_{o \notin (o(1),\ldots,o(k-1))} \tau(s[k-1],o,q)^{\alpha} \cdot \eta(s[k-1],o)^{\beta}} \tag{3.3}$$

$$\tau(s[k-1],v,q) = \left\{ \begin{array}{cc} 1 & \text{si } V_c \text{ está vacío} \\ \frac{\sum_{x \in V_q} M_{xv}}{|V_c|} & \text{si no} \end{array} \right\} \tag{3.4}$$

$$\eta(s[k-1],i) = |W| - deg_W(v) \tag{3.5}$$

De esta forma, el evento ejecutado en cada agente consiste en varios pasos. Primero, se escoge un nodo no coloreado al azar y se le asigna el primer color, representado por 0. Después se escoge un nodo no adyacente con probabilidad $p_{it}(k,v)$, asignándole el mismo color. Este proceso se repite hasta que ya no haya nodos disponibles para ser coloreados con este mismo color, momento en el que se cambia el color y se vuelve a elegir otro nodo no coloreado al azar. Una vez se han coloreado todos los nodos del grafo, se ha obtenido un candidato a solución y ha acabado el evento de este agente.

Algoritmo 5: Pseudocódigo de *ANT_RLF(2,2)*

```
W := V                                              /* Nodos sin colorear */
while there are uncolored vertices:
    B := ∅                  /* Nodos no coloreables con el mismo color que v */
    v := randomChoice(W)
    color(v)
    V_q := v
    while there are uncolored, not adjacent vertices:
        B := B ∪ N_W(v)
        W := W\(N_W(v) ∪ v)
        v := chooseNewVertex(W, p_it(k,v))
        V_q := V_q ∪ v
    W := B ∪ N_W(v)
```

El resultado final se obtiene al acabar las ejecuciones de todos los agentes. Cada vez que un agente construye una nueva solución se compara con la solución que menos colores había usado hasta entonces. Si esta vez se ha obtenido una solución con un menor número de colores, se guarda este candidato a solución en su lugar. Este mismo proceso se realiza con la solución que más repeticiones del mismo color tiene. Si la nueva solución tiene algún color con más repeticiones que la antigua, se guarda como candidato a MIS.

3.2. Estructura de la implementación

La primera cuestión que surge al plantearse la implementación de estos dos algoritmos de enjambre es si existe alguna manera de abstraer su funcionamiento para simplificar su desarrollo. Ambos son algoritmos de enjambre, así que ambos consisten en un sistema de agentes que buscan soluciones y se comunican entre sí. De esta forma, se determina que se puede usar un sistema de clases basado en el paradigma de Programación Orientada a Objetos.

El diseño final de la estructura consiste en el diagrama de clases encontrado en la figura 3.1. A continuación se explicará la finalidad de cada parte del sistema:

- **Node:** para poder representar los grafos que analizaremos, se implementa la clase *Node*, que representa un nodo del grafo. Dentro del nodo se referencian los nodos vecinos, de forma que el grafo completo queda representado únicamente con sus nodos. Además, aquí se almacenará el color del nodo representado para la solución obtenida.

- **Colony:** esta clase representa una colonia, un sistema abstracto formado por agentes. Es una superclase, ya que contiene los elementos básicos que ambos algoritmos usan para construir la colonia de agentes, además de definir las relaciones con la clase *Node* y *Agent*. La colonia guarda la representación del grafo en forma de lista de nodos y los agentes que actuarán sobre ella. Por último, contiene un método para imprimir los enlaces de un grafo, útil a la hora de hacer *debugging*.

- **FrogColony:** las especificaciones de la colonia de ranas, es decir, los datos y métodos necesarios para organizar los agentes, se definen en esta clase. El bucle externo del algoritmo *AntCol* está implementado aquí, llamando luego a un método dentro de cada agente para poder construir una solución satisfactoria. Hereda de *Colony*.

- **AntColony:** en este caso, la clase representa la colonia de hormigas. La funcionalidad de los agentes en *AntCol* posee varias diferencias respecto a los agentes en *FrogCol* como hemos comentado anteriormente, así que parte del peso que en *FrogCol* tiene la colonia se traslada a los agentes. Apate de esto, la colonia contiene la matriz M, que representa las feromonas depositadas a la hora de conseguir soluciones, y maneja su actualización. Hereda de *Colony*.

- **Agent:** esta clase representa el agente básico de un algoritmo de enjambre. Carece de métodos, pero sirve para establecer la relación con la colonia que tienen todos estos agentes.

- **FrogAgent:** las ranas son simuladas con instancias de esta clase. Como se ha descrito anteriormente, se deben crear tantos agentes como nodos en *FrogCol*, así que se define esta relación adicional. Además, cabe destacar que un *FrogAgent* no encuentra una solución global por sí mismo. El agente evalúa una solución local, donde se tiene en cuenta únicamente $N_W(v)$, la vecindad del nodo asociado al agente. Hereda de *Agent*.

- **FrogMessage:** los agentes del algoritmo *FrogCol* no están basados en estigmergía. En cambio, se comunican directamente entre ellos - con sus vecinos, para ser específicos. El mensaje usado para comunicarse está implementado en esta clase, y contiene la información mínima necesaria para llevar a cabo el algoritmo.

- **AntAgent:** esta es la clase que simula las hormigas. Este tipo de agente es mucho más pesado que el agente de *FrogCol*, ya que cada uno de ellos busca completar una solución independientemente de los demás. Cada agente guarda una lista de nodos donde irá acumulando su solución particular, y luego la colonia actualizará la matriz de feromonas en base a estas soluciones. Hereda de *Agent*.

Con esta estructura, resulta mucho más simple implementar un programa que, usando únicamente los métodos de la clase *Colony*, pueda usar los dos algoritmos sin preocuparse del funcionamiento interno de los agentes o los distintos tipos de colonias. Todo se reduciría a crear una instancia de la colonia adecuada, modificando si es necesario los parámetros pasados por defecto a cada tipo, y tener un grafo en el formato usado en esta implementación. Este formato de grafo y su generación será abordada en la siguiente sección.

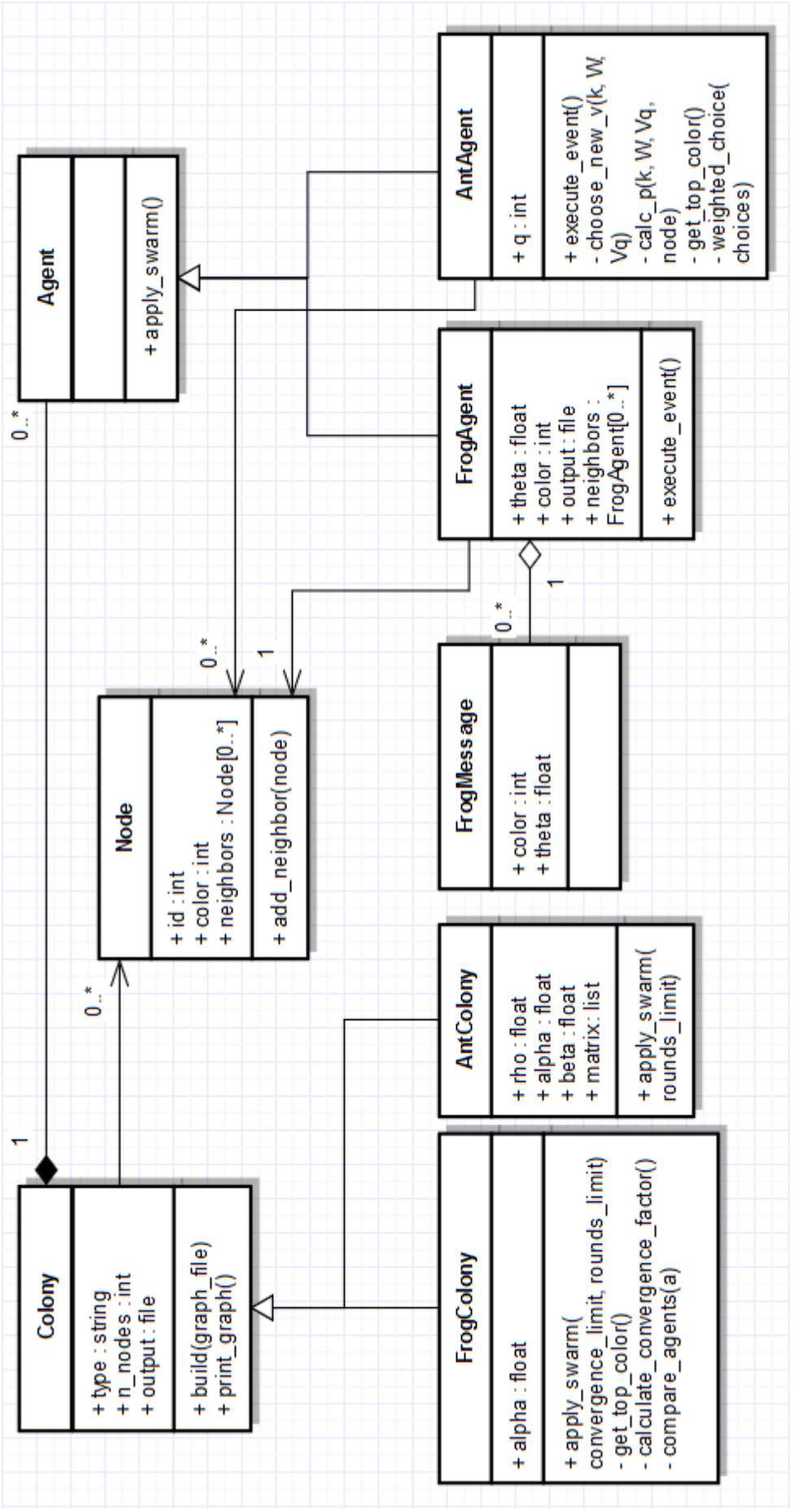

Figura 3.1: Diagrama de clases

3.3. Creación de grafos

Los algoritmos implementados buscan encontrar soluciones al problema de coloreado de grafos, así que para poder probarlos a medida que se fueran desarrollando y hacer una serie de experimentos se necesitaban distintos tipos de grafos. Además, la implementación actual de estos algoritmos únicamente admite grafos dirigidos. Por todo esto, se buscó una forma de obtener rápida y fácilmente grafos no dirigidos sobre los que aplicar *FrogCol* y *AntCol*.

3.3.1. Grafos geométricos aleatorios

Un grafo geométrico aleatorio es un tipo de grafo no dirigido construido mediante la colocación aleatoria de nodos en un espacio métrico. Después, una pareja de nodos colocados de esta manera será unida con un enlace si y solo sí su distancia está dentro de un radio especificado. Fueron introducidos por Gilbert en 1961 [32], sugiriendo su aplicación en redes de comunicación.

Los grafos geométricos aleatorios poseen varias propiedades interesantes. Por ejemplo, desarrollan estructura de comunidades espontáneamente, creando cúmulos con una alta **modularidad**. Además, su estructura hace que posea una alta **asortatividad**, ya que los nodos con muchos enlaces tienden a estar unidos con otros nodos con muchos enlaces. Por todo esto, los grafos geométricos aleatorios se parecen a las comunidades humanas reales.

Estas propiedades, junto con la facilidad a la hora de crearlos, han hecho que se implementase un generador de algoritmos en *python*, que se encuentra en el archivo *swarmgen.py*.

3.3.2. *NetworkX*

Para poder generar los grafos, se ha utilizado la librería *NetworkX*. Este *software*, especialmente diseñada para la creación, manipulación y estudio de grafos y redes, es una librería gratuita que se publicó bajo la licencia *BSD-new* en 2005 [33] y cuya última versión estable data de 20 de septiembre de 2017.

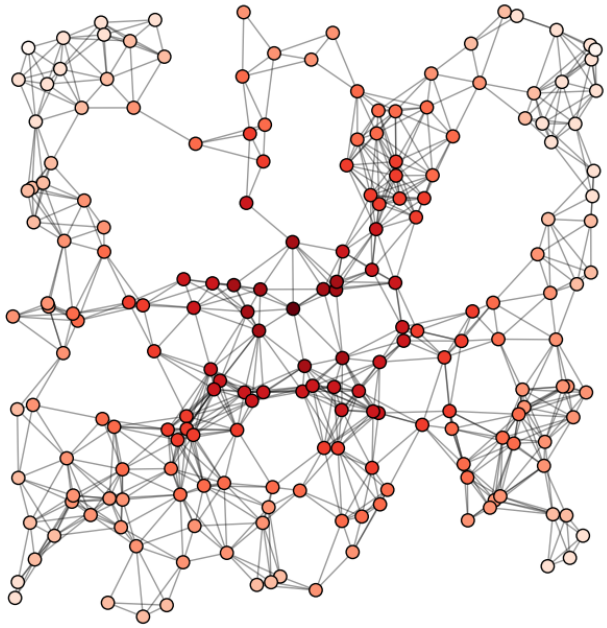

Figura 3.2: Ejemplo de grafo geométrico aleatorio creado con *NetworkX*

Gracias a estar basada en una estructura de diccionarios en *python* nativo, *NetworkX* es una librería eficiente y escalable con la que se puede operar con operaciones de grafos de más de 10 millones de nodos y 100 millones de enlaces [34]. Está incluida en *SageMath* [35]. En este proyecto se ha usado para generar grafos usando su generador de la forma *NetworkX.generators.random_geometric_graph(nodes, radius)*.

4

Experimentos realizados y resultados

Para evaluar el funcionamiento de los algoritmos implementados, se han ejecutado sobre grafos geométricos aleatorios de distintos tamaños y creados con distintos radios. El radio del grafo geométrico determina la distancia a la que dos nodos generan un enlace entre sí, así que los grafos de radio mayor serán más densos y difíciles de procesar. Estos experimentos se han realizado con grafos de 10, 20 y 50 nodos, creados con radio 0,1 y 0,2. Una vez se ha visto la forma en la que la densidad afecta, se han añadido los resultados para grafos de 100, 200, 500, 1000 y 2000 nodos con radio 0,1 en el caso de *FrogCol*; y para grafos de 30 y 40 nodos con radio 0,1 en el caso de *AntCol*

Para obtener unos resultados de la mayor calidad posible, se ha ejecutado 10 veces cada algoritmo sobre el mismo grafo, calculando la media, varianza, valor máximo y valor mínimo de cada resultado analizado. De esta forma se puede ver cómo de consistentes son los algoritmos, además de su calidad puntual.

4.1. *FrogCol*

4.1.1. Tiempos

Nodos	Radio	Tiempo medio	Tiempo varianza	Tiempo máximo	Tiempo mínimo
10	0.1	∼0.0	∼0.0	∼0.0	∼0.0
10	0.2	0.0078	$6.096 \cdot 10^{-5}$	0.016	∼0.0
20	0.1	0.0062	$5.776 \cdot 10^{-5}$	0.016	∼0.0
20	0.2	0.0234	$7.144 \cdot 10^{-5}$	0.033	0.013
50	0.1	0.0515	$5.365 \cdot 10^{-5}$	0.063	0.046
50	0.2	0.0985	$5.065 \cdot 10^{-5}$	0.110	0.093
100	0.1	0.1598	$3.556 \cdot 10^{-5}$	0.172	0.156
200	0.1	0.5934	$4.944 \cdot 10^{-5}$	0.609	0.578
500	0.1	3.9032	0.07233	4.391	3.501
1000	0.1	13.3766	0.42952	14.577	12.671
2000	0.1	53.2222	0.92222	55.641	52.337

Cuadro 4.1: Tabla de tiempos de ejecución de *FrogCol*

En esta tabla se muestran los resultados de tiempo obtenidos por *FrogCol*. Se ha aplicado sobre un conjunto grande de grafos al observar su gran velocidad de ejecución, con una varianza muy pequeña. A continuación se representan los resultados medios en una gráfica, donde se puede apreciar que el tiempo crece de forma exponencial.

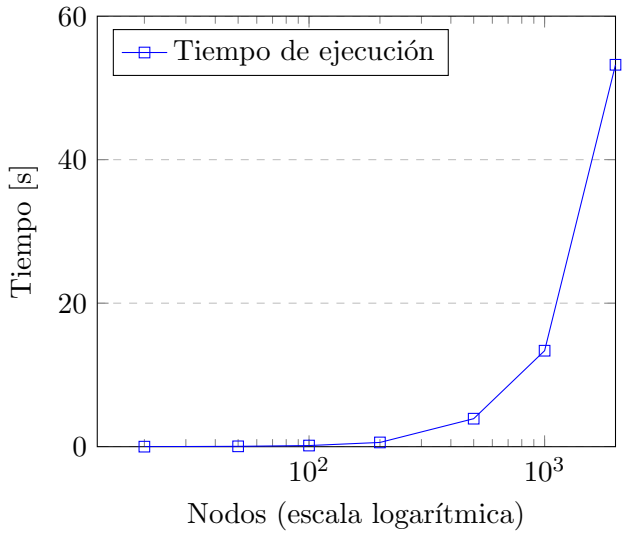

Figura 4.1: Representación de tiempo empleado por *FrogCol* respecto a nº de nodos, $r = 0{,}1$

4.1.2. Coloreados

Esta tabla detalla los coloreados mínimos obtenidos de las ejecuciones. Los resultados obtenidos son muy positivos, porque el número de colores no ha aumentado a pesar de que la cantidad de nodos y enlaces sí que ha variado en gran medida. Además estos coloreados se han conseguido con dos y tres colores, una cantidad muy pequeña.

Nodos	Radio	Coloreado medio	Coloreado varianza	Coloreado máximo	Coloreado mínimo
10	0.1	2.0	0.0	2	2
10	0.2	2.5	0.25	3	2
20	0.1	2.0	0.0	2	2
20	0.2	3.0	0.0	3	3
50	0.1	3.0	0.0	3	3
50	0.2	3.0	0.0	3	3
100	0.1	3.0	0.0	3	3
200	0.1	3.0	0.0	3	3
500	0.1	3.0	0.0	3	3
1000	0.1	3.0	0.0	3	3
2000	0.1	3.0	0.0	3	3

Cuadro 4.2: Tabla de coloreados de *FrogCol*

4.1.3. Grupos maximales independientes

En esta tabla se muestran los resultados de los MIS obtenidos. Cabe destacar que la obtención de estos no era una prioridad a la hora de implementar este algoritmo, pero aún así se han obtenido resultados muy positivos al tener resultados cercanos al 50 % de los grafos coloreados con el mismo color.

Nodos	Radio	MIS medio	MIS varianza	MIS máximo	MIS mínimo
10	0.1	9.0	0.0	9	9
10	0.2	7.4	0.24	8	7
20	0.1	13.5	0.25	14	13
20	0.2	9.7	0.61	11	9
50	0.1	28.3	0.41	29	27
50	0.2	24.0	0.6	25	23
100	0.1	51.9	2.49	55	50
200	0.1	89.8	8.56	94	83
500	0.1	226.5	42.25	240	219
1000	0.1	451.1	58.09	463	438
2000	0.1	894.0	107.2	910	872

Cuadro 4.3: Tabla de obtención de MIS de *FrogCol*

4.2. *AntCol*

4.2.1. Tiempos

Nodos	Radio	Tiempo medio	Tiempo varianza	Tiempo máximo	Tiempo mínimo
10	0.1	0.3591	0.0308	0.641	0.078
10	0.2	0.4375	0.0517	0.797	0.078
20	0.1	3.9173	3.8084	6.922	0.735
20	0.2	4.7033	5.951	8.39	0.86
30	0.1	16.9101	76.5019	30.734	3.42
40	0.1	51.1326	714.6724	93.205	9.437
50	0.1	126.3876	4383.4441	226.519	22.345
50	0.2	128.018	4426.2875	231.521	22.921

Cuadro 4.4: Tabla de tiempos de ejecución de *AntCol*

Esta tabla muestra los tiempos de ejecución obtenidos al aplicar *AntCol*. Estos tiempos son mucho más grandes que al aplicar *FrogCol*, aunque puede que haya alguna ventaja a la hora de aplicar este algoritmo que se muestre en las siguientes tablas. La siguiente gráfica muestra los tiempos de ejecución, y se puede apreciar que estos también crecen de una forma aproximadamente exponencial.

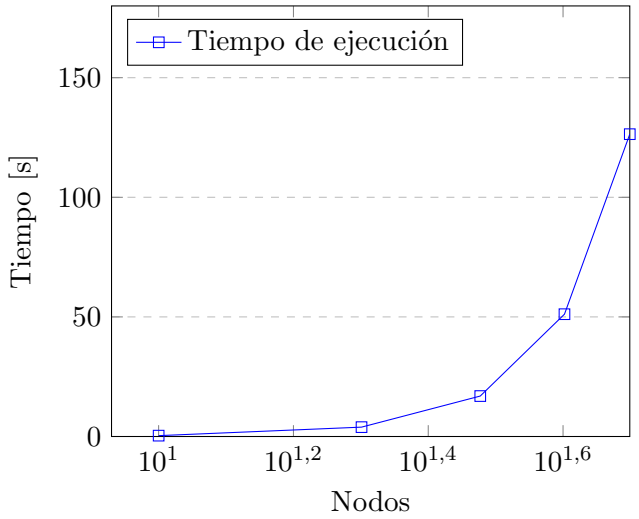

Figura 4.2: Representación de tiempo empleado por *AntCol* respecto a nº de nodos, $r = 0{,}1$

4.2.2. Coloreados

La siguiente tabla muestra los diferentes coloreados mínimos obtenidos de la ejecución del algoritmo. Resulta sorprendente observar que ya desde los grafos de 20 nodos *AntCol* necesita cuatro colores para completar el coloreado, y llega a seis con el grafo de radio 0,2. Esto significa que *FrogCol* es mucho más eficiente y obtiene resultados de mayor calidad que *AntCol*.

Nodos	Radio	Coloreado medio	Coloreado varianza	Coloreado máximo	Coloreado mínimo
10	0.1	2.0	0.0	2	2
10	0.2	3.0	0.0	3	3
20	0.1	2.0	0.0	2	2
20	0.2	4.0	0.0	4	4
30	0.1	3.0	0.0	3	3
40	0.1	3.0	0.0	3	3
50	0.1	4.0	0.0	4	4
50	0.2	6.0	0.0	6	6

Cuadro 4.5: Tabla de coloreados de *AntCol*

4.2.3. Grupos maximales independientes

En la siguiente tabla se muestran los tamaños de los MIS obtenidos al ejecutar *AntCol*. Como en el caso de *FrogCol*, este resultado no ha sido el objetivo principal de la implementación, así que no se han tomado medidas especiales para optimizarlo. Sin embargo, en este caso el algoritmo obtiene unos resultados muy similares a los de *FrogCol*, siendo bastante positivos.

Nodos	Radio	MIS medio	MIS varianza	MIS máximo	MIS mínimo
10	0.1	9.0	0.0	9	9
10	0.2	7.0	0.0	7	7
20	0.1	14.0	0.0	14	14
20	0.2	9.0	0.0	9	9
30	0.1	19.0	0.0	19	19
40	0.1	27.0	0.0	27	27
50	0.1	28.0	0.0	28	28
50	0.2	16.6	0.24	17	16

Cuadro 4.6: Tabla de obtención de MIS de *AntCol*

4.2.4. Número de agentes

Las ejecuciones que se han realizado están hechas con 5 agentes. Como se han obtenido malos resultados tanto en tiempos como en coloreado, se realiza el siguiente experimento, ejecutando el algoritmo con 4 y 3 agentes. Esto sirve para determinar si el tiempo ha sido muy afectado por este número, aunque a la vista de lo obtenido el número de agentes no modifica el tiempo de forma considerable. Por tanto, la conclusión sigue siendo que el algoritmo *FrogCol* es mejor a la hora de colorear y obtener MIS de un grafo.

Agentes	Radio	Tiempo medio	Tiempo varianza	Coloreado medio	MIS medio
5	0.1	3.9173	3.8084	14.0	2.0
5	0.2	4.7033	5.951	9.0	4.0
4	0.1	3.0875	2.5691	14.0	2.0
4	0.2	3.719	3.7872	9.0	4.0
3	0.1	2.3423	1.5923	14.0	2.0
3	0.2	2.9719	2.5246	9.0	4.0

Cuadro 4.7: Tabla comparativa de *AntCol* con distinta cantidad de agentes

4.3. Resultados del experimento

De los resultados obtenidos en estos experimentos se pueden extraer varias conclusiones. La primera de ellas es que el algoritmo *FrogCol* es, en definitiva, un mejor algoritmo para el coloreado de grafos que *AntCol*, ya que requiere de un menor coste computacional y sus resultados son mejores en todas las pruebas realizadas, así que no depende del tamaño del grafo ni de la densidad del mismo. Aparte de esto, se ha observado que la obtención del MIS tiene resultados similares aplicando los dos algoritmos. La única ventaja del algoritmo *AntCol* sobre *FrogCol* radica en la ausencia de varianza en la obtención de estos grupos. Esto signica que una vez hemos obtenido un resultado con él, es muy probable que ese resultado esté cerca del resultado óptimo que se podría obtener con el algoritmo. *FrogCol* fluctúa más, pero el resto de ventajas hace que siga siendo un algoritmo superior en el ámbito global del coloreado de grafos. En las siguientes gráficas mostramos una comparativa de los coloreados y MIS obtenidos con ambos algoritmos. Debido al crecimiento del tiempo empleado en ejecutar *AntCol*, se han tomado valores intermedios en lugar de aumentar el número de nodos.

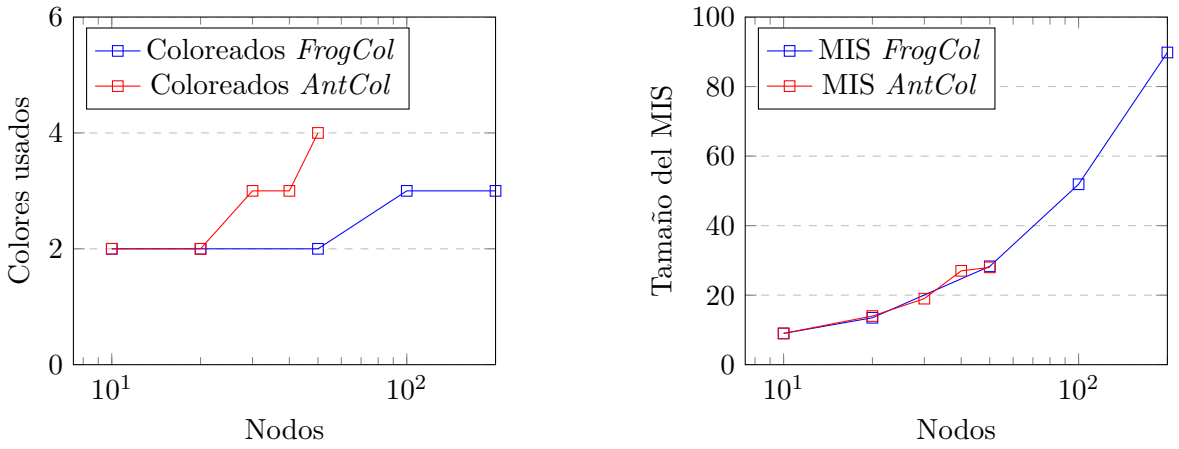

Figura 4.3: Comparativa de coloreados y MIS obtenidos por *FrogCol* y *AntCol*

5

Conclusiones y trabajo futuro

5.1. Conclusiones

La inteligencia de enjambre es un área de estudio que, aunque en los últimos tiempos ha sufrido una gran afluencia de trabajos y estudios de baja calidad, puede seguir aportando innovadores algoritmos con interesantes resultados. Aplicando correctamente la capacidad de abstraer los procesos naturales que tienen lugar en un enjambre, dando lugar a un algoritmo o metaheurística, se pueden diseñar procesos que optimicen o aproximen problemas que de otro modo resultarían muy difíciles de abordar, como los problemas NP-completos.

El comportamiento de las ranas arborícolas japonesas sirvió a Christian Blum de inspiración para la creación del algoritmo *FrogCol*. Este algoritmo, que se diferencia de otros del mismo ámbito en la simplicidad de sus agentes y en el sistema de comunicación no basado en estigmergía, ha sido implementado en *python* de forma que optimice el problema de coloreado de grafos. Este problema es objeto de estudio actual ya que tiene muchas y muy diversas aplicaciones en el mundo real, desde redes *wireless* hasta distribución de turnos de trabajo. Para poder comprobar su rendimiento, al mismo tiempo que se crea una estructura escalable de algoritmos de enjambre, se ha implementado *AntCol*, una versión de la metaheurística ACO.

Una vez han sido implementados, con una estructura de clases y herencias basándose en el paradigma OOP, se han generado una batería de grafos geométricos aleatorios de distintas características sobre los que probarlos. Los resultados obtenidos dejan claro la superioridad del algoritmo *FrogCol* sobre nuestra implementación de *AntCol* en casi todos los ámbitos, destacando su rapidez de ejecución. Esto puede deberse a que el algoritmo *FrogCol* fue diseñado orientado a este tipo de problemas desde el principio, pero no deja de resultar interesante y abre la puerta a que sea abstraído y/o aplicado a más ámbitos.

5.2. Trabajo futuro

Tras analizar los positivos resultados obtenidos, se ha llegado a la conclusión de que *FrogCol* posee mucho potencial. Por ello, a continuación se detallan algunos de los desarrollos a tener en cuenta en el futuro. También la forma de estructurar el código implementado hace posible que sea escalado y ampliado fácilmente.

- **Grafos dirigidos y mixtos:** en estos momentos, la superclase *Colony*, que se encarga de leer los ficheros recibidos y extraer los nodos y enlaces representados en ellos, únicamente trabaja con grafos no dirigidos. Un desarrollo relativamente sencillo sería introducir la posibilidad de analizar grafos dirigidos, modificando la creación del instancias de *Node* y la asignación de vecinos.
- **Ampliar a más algoritmos:** la implementación en forma de clases con herencia hace que el sistema sea flexible y fácilmente escalable. Un desarrollo a tener en cuenta sería añadir más algoritmos de coloreado de grafos, implementando las subclases de *Agent* y *Colony* apropiadas con los métodos usados en las actuales. Un ejemplo sería la implementación de *BeeCol*, basado en el algoritmo ABC, optimización por colonia de abejas artificial; o *PartiCol*, basado en el algoritmo PSO.
- **Abstraer *FrogCol*:** el algoritmo diseñado por Blum, que se ha reimplementado en este proyecto, está pensado únicamente para problemas relacionados con el coloreado de grafos. Un posible desarrollo sería abstraer el funcionamiento del algoritmo, diseñando un pseudocódigo o metaheurística que no dependa del problema a optimizar.

Glosario de acrónimos

- **ABC**: Artificial Bee Colony Optimization, optimización por colonia de abejas artificial
- **ACO**: Ant Colony Optimization, optimización por colonia de hormigas
- **ADN**: Ácido Desoxirribonucleico
- **Boid**: Bird-oid Object
- **GCP**: Graph Coloring Problem, problema de coloreado de grafos
- **MIS**: Maximal Independent Set, grupo máximo independiente
- **NP**: Non-deterministic Polynomial time, tiempo polinómico indeterminado
- **OOP**: Object Oriented Programming, programación orientada a objetos
- **PSO**: Particle Swarm Optimization, optimización por enjambre de partículas
- **RLF**: Recursive Largest First
- **TPS**: Travelling Salesman Problem, problema del viajante

Bibliografía

[1] Gerardo Beni and Wang Jing. Swarm intelligence in cellular robotic systems. *Proceedings of the NATO Advanced Workshop on Robots and Biological Systems*, 1989.

[2] Hugo Hernández and Christian Blum. Distributed graph coloring: an approach based on the calling behavior of japanese tree frogs. *Cornell University Library*, 2010.

[3] David Robinson. The incredible growth of python. *stackoverflow.blog*, 2017.

[4] Christian Blum, María J. Blesa, and Borja Calvo. Frogcol and frogmis: new decentralized algorithms for finding large independent sets in graphs. *Universitat Politècnica de Catalunya*, 2015.

[5] Alain Hertz and D. Costa. Ants can colour graphs. *Journal of the Operational Research Society*, pages 295–305, 1997.

[6] Eric Bonabeu, David Corne, Joshua Knowles, and Riccardo Poli. Swarm intelligence theory: A snapshot of the state of the art. *Theoretical Computer Science*, 2010.

[7] Craig Reynolds. Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, page 25–34, 1987.

[8] Carlos Delgado-Mata, Jesus Ibanez Martinez, Simon Bee, Rocio Ruiz-Rodarte, and Ruth Aylett. On the use of virtual animals with artificial fear in virtual environments. *New Generation Computing. 25*, page 145–169, 2007.

[9] Christopher Hartman and Bedrich Benes. Autonomous boids. *Computer Animation and Virtual Worlds. 17*, page 199–206, 2006.

[10] Martin Saska, Vonasek Vojtech, Krajnik Tomas, and Preucil Libor. Coordination and navigation of heterogeneous uavs-ugvs teams localized by a hawk-eye approach. *International Conference on Intelligent Robots and Systems*, 2012.

[11] Kenneth Sörensen. Metaheuristics - the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[12] Fred Glover and Kenneth Sörensen. Metaheuristics. *Scholarpedia*, 2015.

[13] Marco Dorigo. Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano*, 1992.

[14] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26(1):29–41, 1996.

[15] David Martens, Manu De Backer, Raf Haesen, Jan Vanthienen, Monique Snoeck, and Bart Baesens. Classification with ant colony optimization. *IEEE Transactions on Evolutionary Computation*, 11(5):651–665, 2007.

[16] James Kennedy and Russell C. Eberhart. Particle swarm optimization. *Proceedings of the IEEE International Conference on Neural Networks*, 6, 1995.

[17] Ali Kaveh. Advances in metaheuristic algorithms for optimal design of structures. *Springer*, 2014.

[18] Riccardo Poli. An analysis of publications on particle swarm optimisation applications. *Department of Computer Science, University of Essex*, 2007.

[19] Iztok Jr. Fister, Iztok Fister, Xin-She Yang, Dusan Fister, and Janez Brest. A brief review of nature-inspired algorithms for optimization. *Cornell University Library*, 2013.

[20] Claus Aranha and Felipe Campelo. Evolutionary computation bestiary. *https://github.com/fcampelo/EC-Bestiary*, 2017 (last update).

[21] James Joseph Sylvester. On an application of the new atomic theory to the graphical representation of the invariants and covariants of binary quantics, — with three appendices. *American Journal of Mathematics*, 1:64–104, 1878.

[22] Donald MacKenzie. Mechanizing proof: Computing, risk, and trust. *MIT Press*, page 103, 2004.

[23] Robin Wilson. Four colors suffice. *Princeton University Press*, pages 216–222, 2014.

[24] R. Leonard Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 194–197, 1941.

[25] Richard M. Karp. The complexity of theorem proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, page 151–158, 1971.

[26] A. A. Zykov. On some properties of linear complexes. *American Mathematical Society Translation*, 24:163–188, 1952.

[27] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, page 85–86, 1967.

[28] D. J. Leith and P. Clifford. A self-managed distributed channel selection algorithm for wlan. *Proceedings of the International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, 2006.

[29] K. Duffy, N. O'Connell, and A. Sapozhnikov. Complexity analysis of a decentralised graph colouring algorithm. *Information Processing Letters*, pages 60–63, 2008.

[30] K. D. Wells. The social behaviour of anuran amphibians. *Animal Behaviour*, pages 666–693, 1977.

[31] I. Aihara. Modeling synchronized calling behavior of japanese tree frogs. *Physical Review*, 80:11–18, 2009.

[32] Edgar Gilbert. Random plane networks. *Journal of the Society for Industrial and Applied Mathematics*, 9:533–543, 1961.

[33] Aric Hagberg. Networkx first public release (nx-0.2). *Python-announce-list mailing list*, 2005.

[34] Aric Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. *Proceedings of the 7th Python in Science conference*, 2008.

[35] Sagemath components. *http://www.sagemath.org/links-components.html*.

A

Código Implementado

A.1. Fichero base.py

Este fichero contiene las superclases de las que heredará cada algoritmo implementado. Están representados los agentes básicos y las colonias, junto a los métodos generalizables a todas las subclases.

A.1.1. Node

```
class Node:
    def __init__(self, i):
        self.id = i
        self.neighbors = []
        self.color = i

    def add_neighbor(self, node):
        self.neighbors.append(node)

    def __repr__(self):
        return "N" + str(self.id)
```

A.1.2. Colony

```
class Colony:
    def __init__(self, colony_type, output_file):
        self.type = colony_type
        self.agents = []
        self.n_nodes = 0
        self.nodes = []
```

```
        self.output = output_file

    def build(self, graph_file):
        graph_file = open(graph_file, 'r')

        self.n_nodes = int(graph_file.readline())
        if self.nodes:
            self.nodes = []
        for i in range(0, self.n_nodes):
            self.nodes.append(Node(i))

        for line in islice(graph_file, 0, None):
            arc = line.split()
            self.nodes[int(arc[0])].add_neighbor(self.nodes[int(arc[1])])
            self.nodes[int(arc[1])].add_neighbor(self.nodes[int(arc[0])])

    def print_graph(self):
        self.output.write("---------\n")
        self.output.write("START GRAPH (" + str(self.n_nodes) + ")\n")
        for node in self.nodes:
            line = str(node.id) + "(" + str(node.color) + ") - ["
            for i in xrange(len(node.neighbors)):
                line += str(node.neighbors[i].id)
                if i+1 != len(node.neighbors):
                    line += ", "
            line += "]"
            self.output.write(line + "\n")
        self.output.write("END GRAPH\n")
        self.output.write("---------\n")
```

A.1.3. Agent

```
class Agent:
    def __init__(self, colony):
        self.colony = colony

    def execute_event(self):
        raise NotImplementedError("Agents need execute_event method.")
```

A.2. Fichero frogcol.py

Este fichero contiene las especificaciones del agente y la colonia del algoritmo *FrogCol*, además de la definición de los mensajes usados para la comunicación entre agentes.

A.2.1. FrogColony

```
class FrogColony(base.Colony):
    def __init__(self, output_file, alpha):
        base.Colony.__init__(self, 'frog', output_file)
        self.alpha = alpha        # Variable del usuario

    def build(self, graph_file):
        base.Colony.build(self, graph_file)

        # Creacion de agentes rana, 1 por nodo
        if self.agents:
            self.agents = []
        for node in self.nodes:
            self.agents.append(FrogAgent(self, node, random.uniform(
                                                    0, 1)))

        # Asignacion de ranas vecinas
        for node in self.nodes:
            for neighbor in node.neighbors:
                self.agents[node.id].neighbors.append(
                                                self.agents[neighbor.id])

    # Aplicar algoritmo de enjambre
    def apply_swarm(self, convergence_limit, rounds_limit):
        colors = []
        min_colors = []
        min_colors_used = len(self.nodes)    # Colores usados inicializados

        msize = 0
        rounds_best_found = 0
        rounds_top_found = 0
        # Ordenamos la rana segun theta
        swarm_agents = sorted(self.agents, key=self.compare_agents)
        convergence = 1000
        rounds = 0
        while convergence > convergence_limit and rounds < rounds_limit:
            for agent in swarm_agents:
                # Cada rana colorea su nodo y comunica
                agent.execute_event()

            # Calculo del color mas repetido - MIS
            r = self.get_top_color()
            top_color_n = r[1]
            if top_color_n > msize:
```

```
                colors = r[0]
                msize = top_color_n
                rounds_best_found = rounds

            # Asignacion de coloreado minimo
            colors_used = len(r[0].keys())
            if min_colors_used > colors_used:
                min_colors_used = colors_used
                min_colors = r[0]
                rounds_top_found = rounds

            # Reordenacion de ranas
            swarm_agents = sorted(swarm_agents, key=self.compare_agents)
            convergence = self.calculate_convergence_factor()
            rounds += 1

        return [msize, colors, rounds_top_found, min_colors_used,
                min_colors, rounds_best_found, convergence, rounds,
                self.n_nodes]

    def get_top_color(self):
        colors = {}

        # Guardamos los colores y sus repeticiones
        for agent in self.agents:
            if agent.color in colors:
                colors[agent.color] += 1
            else:
                colors[agent.color] = 1

        # Vemos el color mas repetido
        max_color = 0
        for color in colors.keys():
            if colors[color] > colors[max_color]:
                max_color = color

        return [colors, colors[max_color]]

    @staticmethod
    def compare_agents(a):
        return a.theta

    # Calculo de factor de convergencia
    def calculate_convergence_factor(self):
        new_convergence = 0
        for i in range(0, self.n_nodes - 1):
            if self.agents[i].theta > self.agents[i].theta_old:
                vk = self.agents[i].theta_old
                vg = self.agents[i].theta
            else:
                vg = self.agents[i].theta_old
```

```
                vk = self.agents[i].theta

            dif1 = vg - vk
            dif2 = vk + 1 - vg
            if dif1 < dif2:
                new_convergence += dif1
            else:
                new_convergence += dif2

        new_convergence /= self.n_nodes
        return new_convergence
```

A.2.2. FrogAgent

```
class FrogAgent(base.Agent):
    def __init__(self, colony, node, theta):
        base.Agent.__init__(self, colony)
        self.theta = theta      # Theta para ordenar
        self.theta_old = -1
        self.color = 0
        self.msg_queue = []
        self.neighbors = []
        self.stop = False
        self.node = node

    def execute_event(self):
        self.theta_old = self.theta     # Guardado theta antigua

        # Inicializacion de lista de colores ya usados, True/False para
        # ver si esta cogido
        taken_colors = [False for i in xrange(self.colony.n_nodes)]

        suma = 0
        # Vemos la lista de mensajes
        for msg in self.msg_queue:
            x = msg.theta - self.theta
            x = math.sin(2 * math.pi * x) / (2 * math.pi) # Nuevo valor
            suma += x                           # Sumamos para calcular
                                                # nueva theta
            taken_colors[msg.color] = True      # Color de mensaje asignado

        self.theta -= (self.colony.alpha * suma)
        self.theta = math.modf(self.theta)[0]       # Nueva theta

        i = 0
        if taken_colors:
            while i < len(taken_colors) and taken_colors[i]:
                i += 1
            self.color = i          # Asignamos nuevo color distinto a
```

```
            self.node.color = i      # vecinos y modificamos nodo

        # Creamos mensaje a transmitir
        new_message = FrogMessage(self, self.color, self.theta)

        # Enviamos mensaje a vecinos si no tienen mensaje nuestro ya
        for neighbor in self.neighbors:
            new_queue = []
            for msg in neighbor.msg_queue:
                if msg.sender != self:
                    new_queue.append(msg)
                    break
            new_queue.append(new_message)
            neighbor.msg_queue = new_queue
        # Reiniciamos cola de mensajes
        self.msg_queue = []
```

A.2.3. FrogMessage

```
class FrogMessage:
    def __init__(self, sender, color, theta):
        self.sender = sender
        self.color = color
        self.theta = theta
```

A.3. Fichero antcol.py

Este fichero contiene las especificaciones del agente y la colonia del algoritmo *AntCol*.

A.3.1. AntColony

```
class AntColony(base.Colony):
    def __init__(self, output_file, n_ants):
        base.Colony.__init__(self, 'ant', output_file)
        self.n_ants = n_ants     # Numero de hormigas

        self.rho = 0.5           # Variables del usuario
        self.alpha = 2           # - mejores resultados cuando
        self.beta = 3            #   rho=0.5, alpha=2, beta in [2,4]

        self.matrix = []         # Matriz que representa la "eficiencia"
                                 # del camino entre nodos i y j en M[i][j]

    # Inicializacion de la colonia
    def build(self, graph_file):
        base.Colony.build(self, graph_file)
```

```
        # Inicializacion matriz de rastros a 0
        self.matrix = [[0 for x in range(self.n_nodes)] for y
                       in range(self.n_nodes)]

        # Creacion de agentes hormiga
        for i in range(0, self.n_ants):
            self.agents.append(AntAgent(self))

        # Inicializacion de matriz de rastros a 1 cuando i y j no el
        # mismo ni vecinos
        for i in range(0, self.n_nodes):
            for j in range(0, self.n_nodes):
                node1 = self.nodes[i]
                node2 = self.nodes[j]
                if node2 not in node1.neighbors and node1 != node2:
                    self.matrix[i][j] = 1

    # Aplicar algoritmo de enjambre // Basicamente Table2 del paper
    def apply_swarm(self, convergence_limit, rounds_limit):
        colors = []
        min_colors = []
        min_colors_used = len(self.nodes)   # Colores usados iniciali-
                                            # zados al maximo
        msize = 0
        rounds_best_found = 0
        rounds_top_found = 0
        self.matrix = [[1 for x in range(self.n_nodes)] for y
                       in range(self.n_nodes)]


        rounds = 0
        while rounds < rounds_limit:
            # Inicializacion de matriz dM, contendra variacion de M
            var_matrix = [[0 for x in range(self.n_nodes)] for y
                          in range(self.n_nodes)]

            for agent in self.agents:
                # Cada hormiga ordena y colorea el grafo
                agent.execute_event()

                # Calculo del color mas repetido - MIS
                r = agent.get_top_color()
                top_color_n = r[1]
                if top_color_n > msize:
                    colors = r[0]
                    msize = top_color_n
                    rounds_best_found = rounds

                # Asignacion de coloreado minimo
```

```
                colors_used = agent.q
                if min_colors_used > colors_used:
                    min_colors_used = colors_used
                    min_colors = r[0]
                    rounds_top_found = rounds

                # Actualizacion de matriz dM
                var_matrix += [[1 / agent.q for j in xrange(len(
                                agent.aux_nodes)) if
                                agent.aux_nodes[i] != agent.aux_nodes[j]
                                and
                                gent.aux_nodes[i].color ==
                                agent.aux_nodes[j].color]
                                for i in xrange(len(agent.aux_nodes))]

            # Una vez todas las hormigas colorean, se crea nueva M
            for i in range(0, len(self.nodes)):
                for j in range(0, len(self.nodes)):
                    node1 = self.nodes[i]
                    node2 = self.nodes[j]
                    if node2 not in node1.neighbors and node1 != node2:
                        # Nuevo valor de M basado en rho y el valor de dM
                        self.matrix[i][j] = self.rho * self.matrix[i][j]
                                            + var_matrix[i][j]

            rounds += 1

        return [msize, colors, rounds_top_found, min_colors_used,
                min_colors, rounds_best_found, "N/A", rounds,
                self.n_nodes]
```

A.3.2. AntAgent

```
class AntAgent(base.Agent):
    def __init__(self, colony):
        base.Agent.__init__(self, colony)
        self.q = 0                  # Numero de colores usados en iteracion
        self.aux_nodes = []         # Lista de nodos auxiliar

    # Ordena el grafo dinamicamente mientras lo colorea // ANT_RLF(2, 2)
    def execute_event(self):
        q = 0                               # Numero de colores usados
        W = list(self.colony.nodes)         # Copia de la lista de nodos
        k = 0                               # Numero de vertices coloreados
        self.aux_nodes = []


        # Mientras no esten todos los vertices coloreados
        while k < self.colony.n_nodes:
```

```
                k += 1                      # Coloreamos nodo
                B = []                      # Inicializacion lista de vertices
                                            # que no se pueden colorear
                v = random.choice(W)        # Sigma = 2
                v.color = q                 # Coloreamos nodo
                self.aux_nodes.append(v)

                Vq = [v]                    # Inicializacion lista vertices
                                            # coloreados

                # W\(Nw(v)U{v}) ->  vertices "coloreables"
                to_color = [node for node in W if node not in v.neighbors
                            and node != v]


                # Para cada nodo "coloreable"
                while to_color:
                    for node in v.neighbors:    # Anyadimos vecinos a B
                        B.append(node)

                    W = to_color                # Posibles nodos a elegir
                                                # ahora "coloreables"
                    v = self.choose_new_v(k, W, Vq)
                    k += 1
                    v.color = q                 # Coloreamos nodo
                    self.aux_nodes.append(v)
                    Vq.append(v)                # Anyadimos nodo a coloreados

                    # W\(Nw(v)U{v}) -> vertices "coloreables"
                    to_color = [node for node in W if node not in v.neighbors
                                and node != v]

                # Una vez coloreados todos los nodos posibles con q, lista
                # de seleccionables es ahora B y los vecinos del ultimo v
                W = [node for node in self.colony.nodes if (node in
                     v.neighbors or node in B) and node not in self.aux_nodes]

                # Tomamos nuevo color
                q += 1

        self.q = q

    # Escoger nuevo vertice
    def choose_new_v(self, k, W, Vq):
        choices = []            # Inicializacion mapa de posibilidades
        for node in W:          # Para cada nodo posible, anyadir con prob
            prob = self.calc_p(k, W, Vq, node)
            choices.append((node, prob))


        # Devolver nodo aleatorio segun pesos
```

```
            return self.weighted_choice(choices)

        # Calcular probabilidad // Pagina 4 del paper
        def calc_p(self, k, W, Vq, node):

            # CALCULO DE T1(s[k-1],v) = T2(s[k-1],v,q)
            if k != 0:
                # T2(s[k-1],v,c) = sum{Miv}/|Vc|
                tau_node = 0
                for i in xrange(self.colony.n_nodes):
                    tau_node += self.colony.matrix[i][node.id]
                tau_node /= k
            else:
                # T2(s[k-1],v,c) = 1
                tau_node = 1

            # CALCULO DE NU1(s[k-1],v) = |W| - degw(v)
            # Lista de vecinos
            W_neighbors = [n_node for n_node in W if n_node in node.neighbors]

            # NU1(s[k-1],v) = |W| - degw(v) = |W| - |Nw(v)|
            deg_node = len(W) - len(W_neighbors)

            # CALCULO DE NUMERADOR DE P = T1^alpha * NU1^beta
            num = 1.0*pow(tau_node, self.colony.alpha)
                  * pow(deg_node, self.colony.beta)

            denom = 0.0
            # Los o son vertices ya coloreados
            for q_node in Vq:
                # CALCULO DE T1(s[k-1],o) = T2(s[k-1],o,q)
                if k != 0:
                    # T2(s[k-1],o,c) = sum{Miv}/|Vc|
                    tau_os = 0
                    for i in xrange(self.colony.n_nodes):
                        tau_os += self.colony.matrix[i][q_node.id]
                    tau_os /= k
                else:
                    # T2(s[k-1],o,c) = 1
                    tau_os = 1

                # CALCULO DE NU1(s[k-1],o) = |W| - degw(o)
                # Lista de vecinos
                W_neighbors = [n_node for n_node in W if n_node
                               in q_node.neighbors]

                # NU1(s[k-1],o) = |W| - degw(o) = |W| - |Nw(o)|
                deg_os = len(W) - len(W_neighbors)

                # CALCULO DE DENOMINADOR DE P = sum_o{T1^alpha * NU1^beta}
                denom += pow(tau_os, self.colony.alpha)
```

```
                    * pow(deg_os, self.colony.beta)

        # Retorno de p
        return num / denom

    def get_top_color(self):
        colors = {}

        # Guardamos los colores y sus repeticiones
        for node in self.aux_nodes:
            if node.color in colors:
                colors[node.color] += 1
            else:
                colors[node.color] = 1

        # Vemos el color mas repetido
        max_color = 0
        for color in colors.keys():
            if colors[color] > colors[max_color]:
                max_color = color

        return [colors, colors[max_color]]

    # Funcion de eleccion con pesos
    @staticmethod
    def weighted_choice(choices):
        values, weights = zip(*choices)
        total = 0
        cum_weights = []
        for w in weights:
            total += w
            cum_weights.append(total)
        x = random.random() * total
        i = bisect(cum_weights, x)
        return values[i]
```

A.4. Fichero graphgen.py

Este fichero contiene el código implementado para generar grafos geométricos aleatorios usando *NetworkX*. Está pensado para poder ser usado desde terminal.

```
radius = 0.1
nodes = 100
write_number = True
filename = "test.txt"

if len(sys.argv) > 2:
    for i in range(1, len(sys.argv)):
        if sys.argv[i] == '-n' or sys.argv[i] == '-nodes':
            nodes = sys.argv[i+1]
        if sys.argv[i] == '-r' or sys.argv[i] == '-radius':
            radius = sys.argv[i+1]
        if sys.argv[i] == '-f' or sys.argv[i] == '-filename':
            filename = sys.argv[i+1]
        if sys.argv[i] == '-wn' or sys.argv[i] == '-writenumber':
            write_number = True

graph = nx.generators.geometric.random_geometric_graph(nodes, radius)

file_object = open(filename, 'w')
if write_number:
    file_object.write(str(graph.order()) + "\n")
nx.write_edgelist(graph, file_object, data=False)
```